

Trabajo 3

Antonio Álvarez Caballero

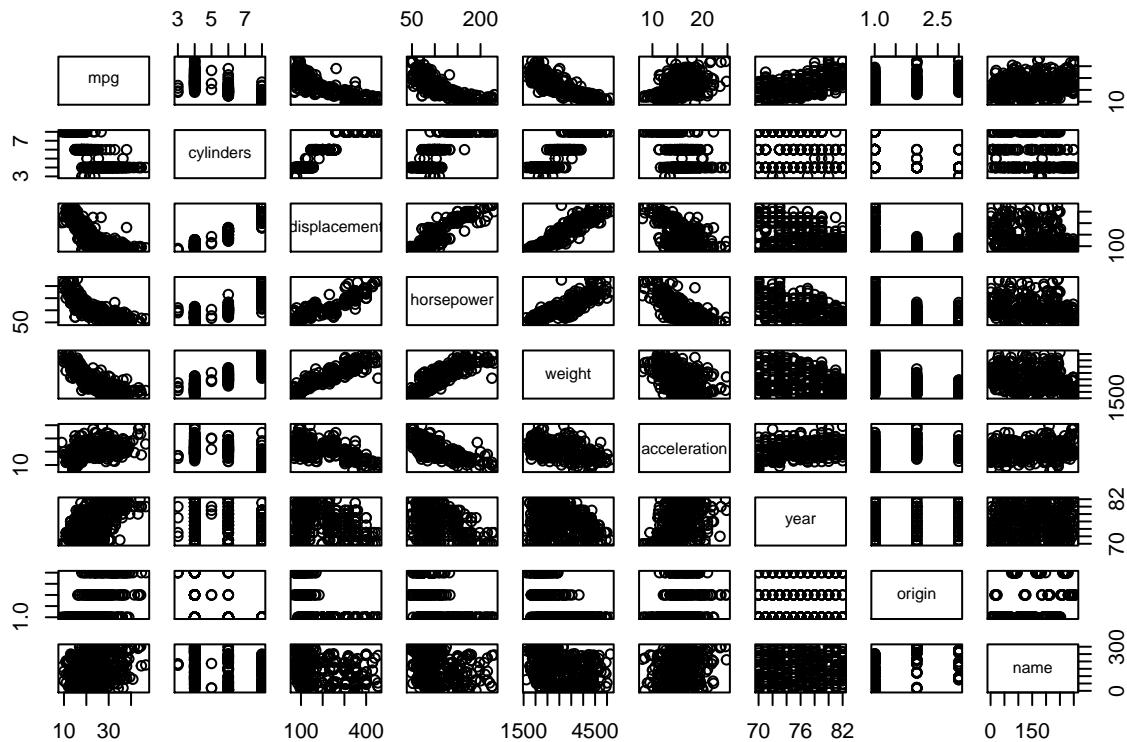
27 de mayo de 2016

Ejercicio 1

```
## The following object is masked from package:ggplot2:
```

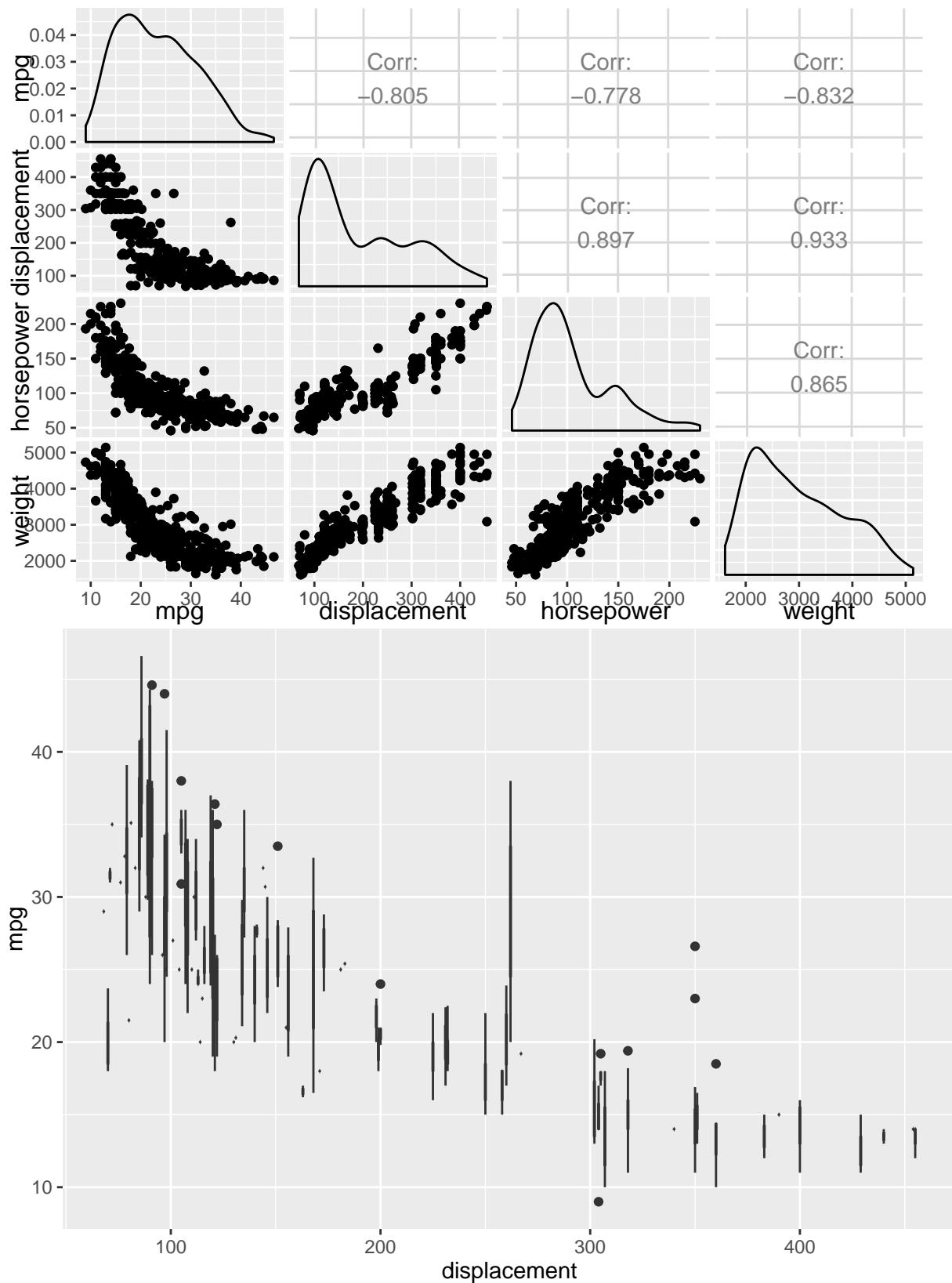
```
##
```

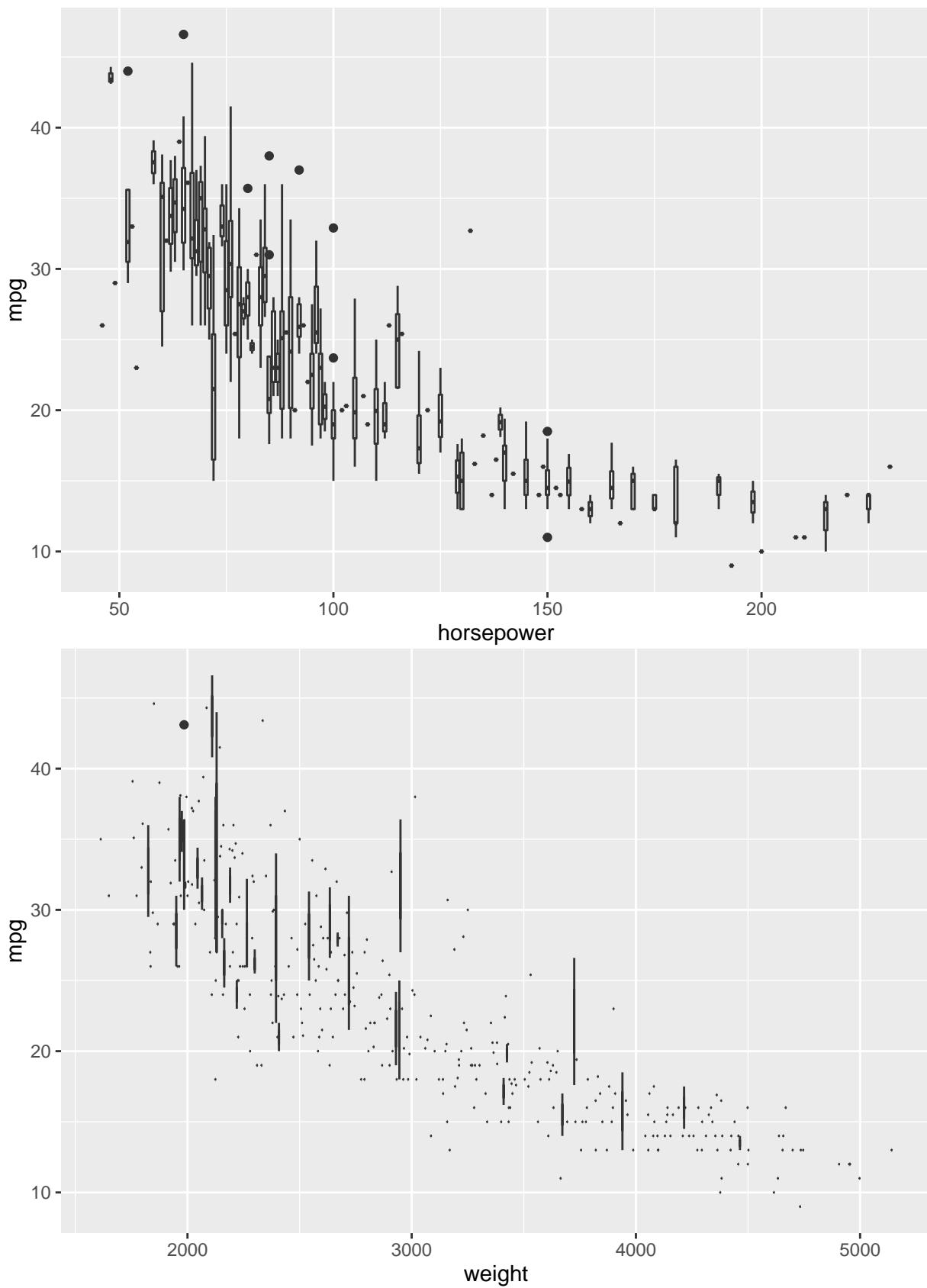
```
##     mpg
```



Apartado a)

Parece ser que las variables de las que más depende *mpg* son *displacement*, *horsepower* y *weight*. Veámoslas con más detalle.





Es claro que existe una dependencia entre estas variables. Además, con el primer plot podemos ver las

correlaciones entre estas 3 variables, que es alta.

Apartado b)

Seleccionamos las variables que hemos decidido para predecir.

```
Auto.selected <- Auto[,c("displacement","horsepower","weight")]
```

Apartado c)

Como nuestro conjunto de datos es grande (392 instancias), podemos realizar un muestreo aleatorio. Así tampoco falseamos las muestras, cosa que podría pasarnos si realizamos un muestreo estratificado.

```
index <- sample(nrow(Auto), size = 0.8*nrow(Auto) )  
  
Auto.train <- Auto.selected[index,]  
Auto.test <- Auto.selected[-index,]
```

Apartado d)

Vamos a crear una nueva variable, *mpg01*, la cual tendrá 1 si el valor de *mpg* está por encima de la mediana y -1 en otro caso.

```
mpg01 <- ifelse(Auto$mpg >= median(Auto$mpg), 1, 0)  
Auto.selected$mpg01 <- mpg01  
Auto.train$mpg01 <- mpg01[index]  
Auto.test$mpg01 <- mpg01[-index]
```

Apartado d1)

Vamos a ajustar un modelo de regresión logística para predecir *mpg01*.

```
model.LogReg <- glm(mpg01 ~ ., data = Auto.train, family = binomial)  
prediction.LogReg <- predict(model.LogReg, newdata = Auto.test, type = "response")  
prediction.LogReg.labels <- ifelse(prediction.LogReg > 0.5, 1, 0)  
error.test <- sum(sign(prediction.LogReg.labels) != sign(Auto.test$mpg01)) /  
length(prediction.LogReg.labels)
```

El error de test de este modelo es 16.4556962.

Apartado d2)

Ahora vamos a ajustar un modelo k-NN.

```
scale.train <- scale(Auto.train[, colnames(Auto.train) != "mpg01"])  
means <- attr(scale.train, "scaled:center")  
scales <- attr(scale.train, "scaled:scale")  
  
scale.test <- scale(Auto.test[, colnames(Auto.train) != "mpg01"], means, scales)  
scale.data <- rbind(scale.train, scale.test)  
scale.labels <- as.factor(c(Auto.train$mpg01, Auto.test$mpg01))
```

```

set.seed(100000004)
tune.result <- tune.knn(scale.data, scale.labels, k = 1:30, tunecontrol = tune.control(sampling = "cross"))

best.k <- tune.result$best.model$k

prediction.knn <- knn(scale.train, scale.test, Auto.train$mpg01, k = best.k, prob = TRUE)
error.test.knn <- sum(prediction.knn != Auto.test$mpg01) / length(prediction.knn)

```

El error del k-NN con $k = 9$ es 12.6582278.

Apartado d3)

Veamos las curvas ROC de ambos modelos.

```

roc.prediction.regLog <- prediction(prediction.LogReg, Auto.test$mpg01)
roc.performance.regLog <- performance(roc.prediction.regLog, measure = "tpr", x.measure = "fpr")

## kNN
prob <- attr(prediction.knn, "prob")
prob <- ifelse(prediction.knn == "0", 1-prob, prob)
roc.prediction.knn <- prediction(prob, Auto.test$mpg01)
roc.performance.knn <- performance(roc.prediction.knn, measure = "tpr", x.measure = "fpr")
####

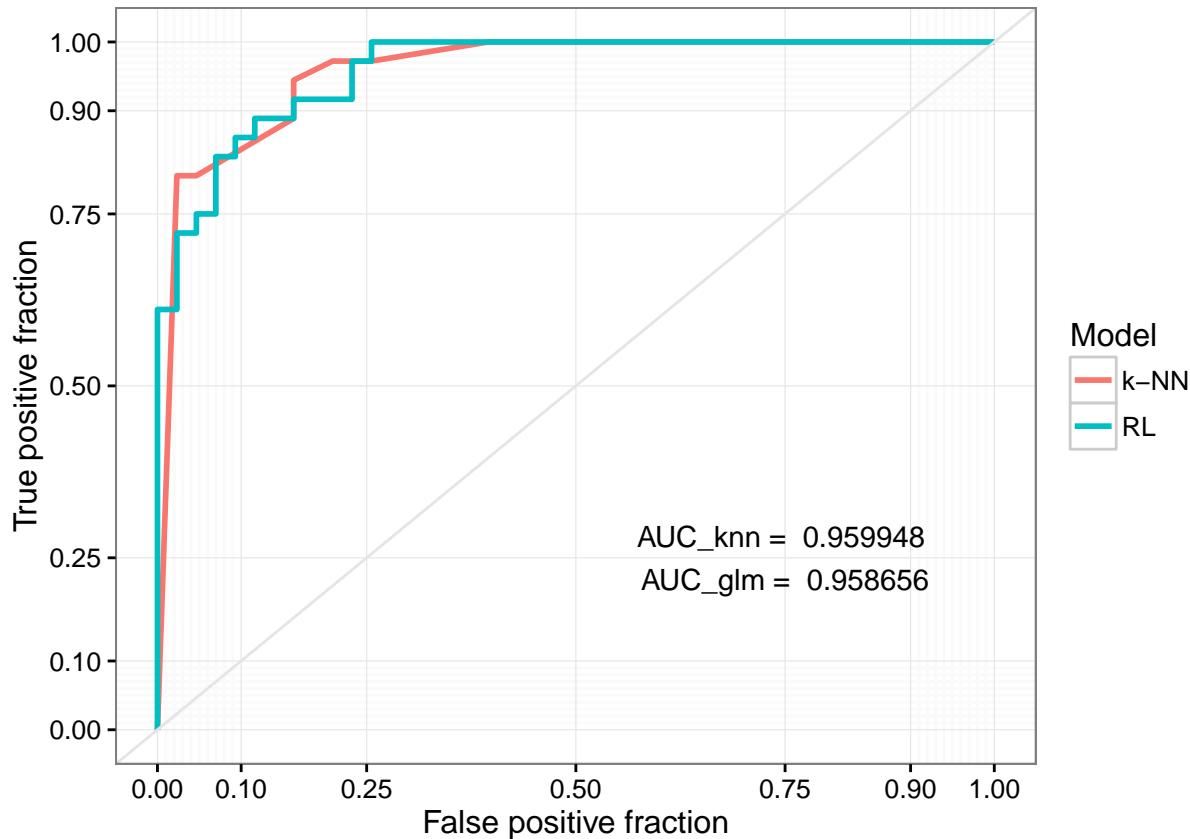
roc.data <- data.frame(M = c(unlist(attr(roc.prediction.knn, "predictions")),
                               unlist(attr(roc.prediction.regLog, "predictions"))),
                         D = c(unlist(attr(roc.prediction.knn, "labels")),
                               unlist(attr(roc.prediction.regLog, "labels"))),
                         Model = c(rep("k-NN", nrow(Auto.test)), rep("RL", nrow(Auto.test)))))

g <- ggplot(roc.data, aes(d = D, m = M, color = Model)) + geom_roc(n.cuts = 0) + style_roc()
auc <- round(calc_auc(g), 6)

## Warning in verify_d(data$d): D not labeled 0/1, assuming 1 = 0 and 2 = 1!
g <- g + annotate("text", x = .75, y = .25,
                  label = paste("AUC_knn = ", auc$AUC[2], "\nAUC_glm = ", auc$AUC[1]))
g

## Warning in verify_d(data$d): D not labeled 0/1, assuming 1 = 0 and 2 = 1!

```



```
auc.regLog <- auc$AUC[1]
auc.knn <- auc$AUC[2]
```

El área bajo la curva de la ROC de regresión logística es 0.958656 y la del k-NN es 0.959948. Luego k-NN es el modelo que mejor *performance* tiene.

Apartado e) (Bonus-1)

Para estudiar el error con validación cruzada hacemos uso de `cv.glm`

```
model.full.LogReg <- glm(mpg01 ~ ., data = Auto.selected)
cv.LogReg <- cv.glm(data = Auto.selected, glmfit = model.full.LogReg, K = 5)
cv.LogReg$delta
```

```
## [1] 0.1030966 0.1029422
```

El error estimado es el primero de este vector. El segundo es un ajuste para compensar el sesgo introducido al no usar *Leave-One-Out*.

Para el caso del k-NN

```
set.seed(1000000004)
prediction.knn.cv <- knn.cv(scale.data, scale.labels, k = best.k, l = 0, prob = FALSE, use.all = TRUE)
error.knn.cv <- sum(prediction.knn.cv != scale.labels) / length(prediction.knn.cv)
error.knn.cv
```

```
## [1] 0.08418367
```

Por tanto, vemos que es mejor k-NN.

Apartado f) (Bonus-2)

Por hacer

Ejercicio 2

Apartado a)

Ajustamos sobre la variable *crim*, que es la que está en la posición 1, con partición de datos de 80% y test de 20% con muestreo aleatorio.

```
attach(Boston)

set.seed(100000004)
index <- sample(nrow(Boston), 0.8*nrow(Boston))
Boston.full <- Boston
Boston.train <- Boston[index,]
Boston.test <- Boston[-index,]

model.Boston <- glmnet(as.matrix(Boston.train[,-1]), Boston.train[,1], alpha = 1)
```

Ahora utilizamos un método LASSO y seleccionamos las variables que están por encima de un umbral.

```
cv.Boston <- cv.glmnet(as.matrix(Boston[,-1]), Boston[,1], nfolds = 5, alpha = 1)
lambda.min <- cv.Boston$lambda.min
lasso.coef <- predict(cv.Boston, type="coefficients", s = lambda.min)
threshold <- 0.4
selected <- which(abs(lasso.coef) > threshold)[-1]
```

Con esto afirmamos que las características que superan nuestro umbral 0.4 son 4, 5, 8, 9. Disminuir este umbral nos haría coger más características.

Apartado b)

Para realizar regularización *weight-decay* debemos usar de nuevo *glmnet* pero con el parámetro $\alpha = 0$.

```
Boston.selected.train <- Boston.train[,selected]
Boston.selected.test <- Boston.test[,selected]

regularized.model <- glmnet(as.matrix(Boston.selected.train),
                             as.matrix(Boston.train[,1]),
                             alpha = 0, lambda = lambda.min)

regularized.prediction <- predict(regularized.model, s = lambda.min,
                                    newx = as.matrix(Boston.selected.test))
```

El error residual es la raíz cuadrada positiva del error cuadrático medio.

```
regularized.residual <- sqrt(mse(Boston.test[,1],regularized.prediction))
```

El error residual es 7.3567959.

Para ver si hay underfitting, probemos con distintos valores de lambda.

```

lambda2 <- 10*lambda.min
regularized.lambda2.model <- glmnet(as.matrix(Boston.selected.train),
                                     as.matrix(Boston.train[,1]),
                                     alpha = 0, lambda = lambda2)

regularized.lambda2.prediction <- predict(regularized.lambda2.model, s = lambda2,
                                             newx = as.matrix(Boston.selected.test))

regularized.lambda2.residual <- sqrt(mse(Boston.test[,1],regularized.lambda2.prediction))

```

El error residual es 7.3597139.

```

lambda3 <- 100*lambda.min
regularized.lambda3.model <- glmnet(as.matrix(Boston.selected.train),
                                     as.matrix(Boston.train[,1]),
                                     alpha = 0, lambda = lambda3)

regularized.lambda3.prediction <- predict(regularized.lambda2.model, s = lambda3,
                                             newx = as.matrix(Boston.selected.test))

regularized.lambda3.residual <- sqrt(mse(Boston.test[,1],regularized.lambda3.prediction))

```

El error residual es 7.3597139.

Hemos visto que al aumentar λ el error residual ha bajado (aunque de 10λ a 100λ no se ha movido, por lo que parece haberse estabilizado), luego podemos afirmar que hay un poco de *underfitting* con el mínimo λ calculado.

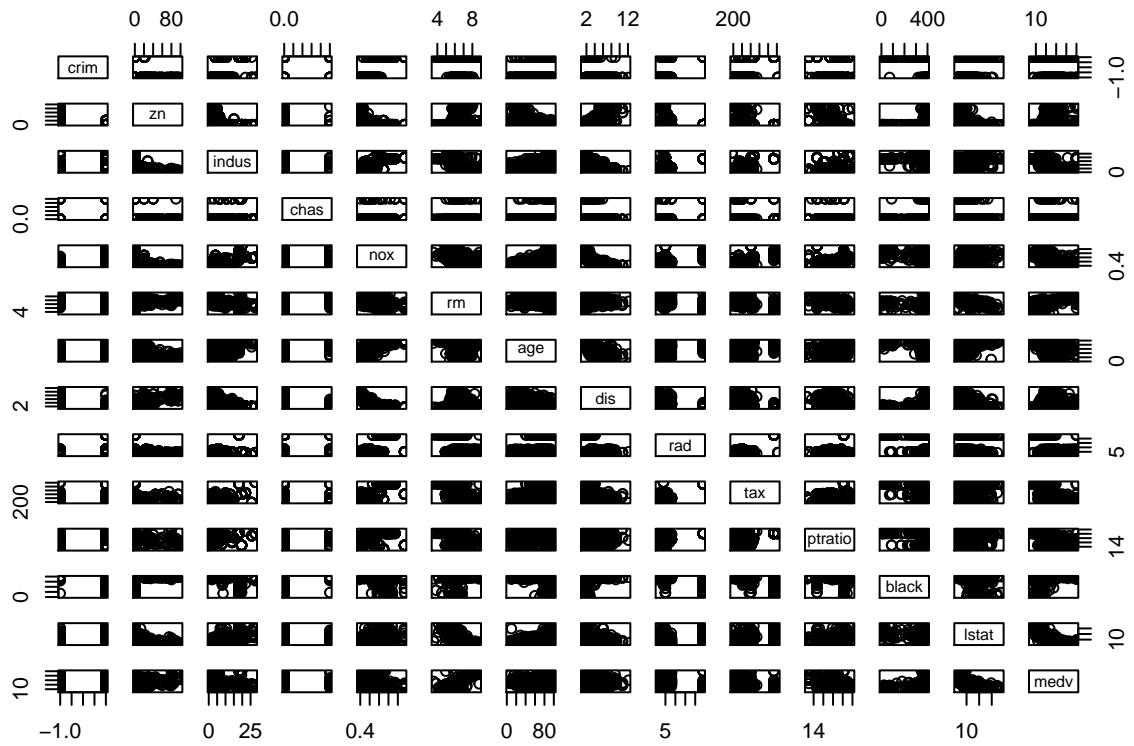
Apartado c)

Al igual que en el anterior apartado, definimos una nueva variable usando la mediana como umbral.

```

Boston.full$crim <- ifelse(Boston$crim > median(Boston$crim), 1, -1)
Boston.train.crim1 <- Boston.full[index,]
Boston.test.crim1 <- Boston.full[-index,]
pairs(Boston.full)

```



Ahora ajustamos varias *SVM*, probaremos la lineal y con los núcleos disponibles, y veremos cómo se comporta cada uno. Posiblemente el núcleo lineal no sea suficiente, ya que los datos no parecen lo suficientemente separados. A priori parece que uno polinomial hará un mejor trabajo.

```
svm.linear <- svm(crim ~ ., data = Boston.train.crim1, kernel = "linear")
svm.linear.prediction <- predict(svm.linear, newdata = Boston.test.crim1)
confusionMatrix(sign(svm.linear.prediction), Boston.test.crim1$crim)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction -1  1
##           -1 53 12
##            1  3 34
##
##                  Accuracy : 0.8529
##                  95% CI : (0.7691, 0.9153)
##      No Information Rate : 0.549
##      P-Value [Acc > NIR] : 6.28e-11
##
##                  Kappa : 0.6977
## Mcnemar's Test P-Value : 0.03887
##
##                  Sensitivity : 0.9464
##                  Specificity : 0.7391
##      Pos Pred Value : 0.8154
##      Neg Pred Value : 0.9189
##                  Prevalence : 0.5490
##      Detection Rate : 0.5196
## Detection Prevalence : 0.6373
##      Balanced Accuracy : 0.8428
```

```

##          'Positive' Class : -1
##
svm.polynomial <- svm(crim ~ ., data = Boston.train.crim1, kernel = "polynomial")
svm.polynomial.prediction <- predict(svm.polynomial, newdata = Boston.test.crim1)
confusionMatrix(sign(svm.polynomial.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##          Reference
## Prediction -1  1
##           -1 56  9
##            1   0 37
##
##          Accuracy : 0.9118
##             95% CI : (0.8391, 0.9589)
## No Information Rate : 0.549
## P-Value [Acc > NIR] : 1.208e-15
##
##          Kappa : 0.8186
## Mcnemar's Test P-Value : 0.007661
##
##          Sensitivity : 1.0000
##          Specificity : 0.8043
## Pos Pred Value : 0.8615
## Neg Pred Value : 1.0000
##          Prevalence : 0.5490
## Detection Rate : 0.5490
## Detection Prevalence : 0.6373
## Balanced Accuracy : 0.9022
##
##          'Positive' Class : -1
##

svm.radial <- svm(crim ~ ., data = Boston.train.crim1, kernel = "radial")
svm.radial.prediction <- predict(svm.radial, newdata = Boston.test.crim1)
confusionMatrix(sign(svm.radial.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##          Reference
## Prediction -1  1
##           -1 51  4
##            1   5 42
##
##          Accuracy : 0.9118
##             95% CI : (0.8391, 0.9589)
## No Information Rate : 0.549
## P-Value [Acc > NIR] : 1.208e-15
##
##          Kappa : 0.8222
## Mcnemar's Test P-Value : 1
##
##          Sensitivity : 0.9107

```

```

##          Specificity : 0.9130
##      Pos Pred Value : 0.9273
##      Neg Pred Value : 0.8936
##          Prevalence : 0.5490
##      Detection Rate : 0.5000
## Detection Prevalence : 0.5392
##      Balanced Accuracy : 0.9119
##
##      'Positive' Class : -1
##
svm.sigmoid <- svm(crim ~ ., data = Boston.train.crim1, kernel = "sigmoid")
svm.sigmoid.prediction <- predict(svm.sigmoid, newdata = Boston.test.crim1)
confusionMatrix(sign(svm.sigmoid.prediction), Boston.test.crim1$crim)

## Confusion Matrix and Statistics
##
##      Reference
## Prediction -1  1
##      -1 43 21
##      1 13 25
##
##          Accuracy : 0.6667
##      95% CI : (0.5664, 0.7569)
##      No Information Rate : 0.549
##      P-Value [Acc > NIR] : 0.01042
##
##          Kappa : 0.3162
##  Mcnemar's Test P-Value : 0.22995
##
##      Sensitivity : 0.7679
##      Specificity : 0.5435
##      Pos Pred Value : 0.6719
##      Neg Pred Value : 0.6579
##          Prevalence : 0.5490
##      Detection Rate : 0.4216
## Detection Prevalence : 0.6275
##      Balanced Accuracy : 0.6557
##
##      'Positive' Class : -1
##

```

Al final, el kernel polinomial ha sido el que mejor ha funcionado. El kernel lineal se comporta muy bien, pero el radial y el polinomial se comportan mejor.

Apartado d) (Bonus-3)

Ajustamos con validación cruzada sobre la variable *crim*.

```

cv.Boston <- cv.glmnet(as.matrix(Boston.full[,-1]), Boston.full[,1], nfolds = 5, alpha = 1,
type.measure="mse")

```

El error de validación cruzada es

```

summary(cv.Boston)
cv.Boston$cvm
mean(cv.Boston$cvm)
prediction.Boston <- predict(cv.Boston, type="class", s = lambda.min)
prediction.Boston

```

Este es el error cuadrático medio por validación cruzada del modelo.

Ejercicio 3

Apartado a)

Ya tenemos cargado y separado el conjunto de datos en 80% training y 20% test.

Apartado b)

Vamos a ajustar un modelo de Bagging. Para ello usaremos el RandomForest y le diremos que use el total de características disponibles. El error de test lo mediremos siempre con el *MSE*.

```

bagging <- randomForest(medv ~., data = Boston, subset = index, mtry = ncol(Boston)-1, importance = TRUE)
bagging.prediction <- predict(bagging, newdata = Boston.test)
bagging.error <- mse(medv[-index], bagging.prediction)

```

El error de test del modelo bagging es 11.5488712.

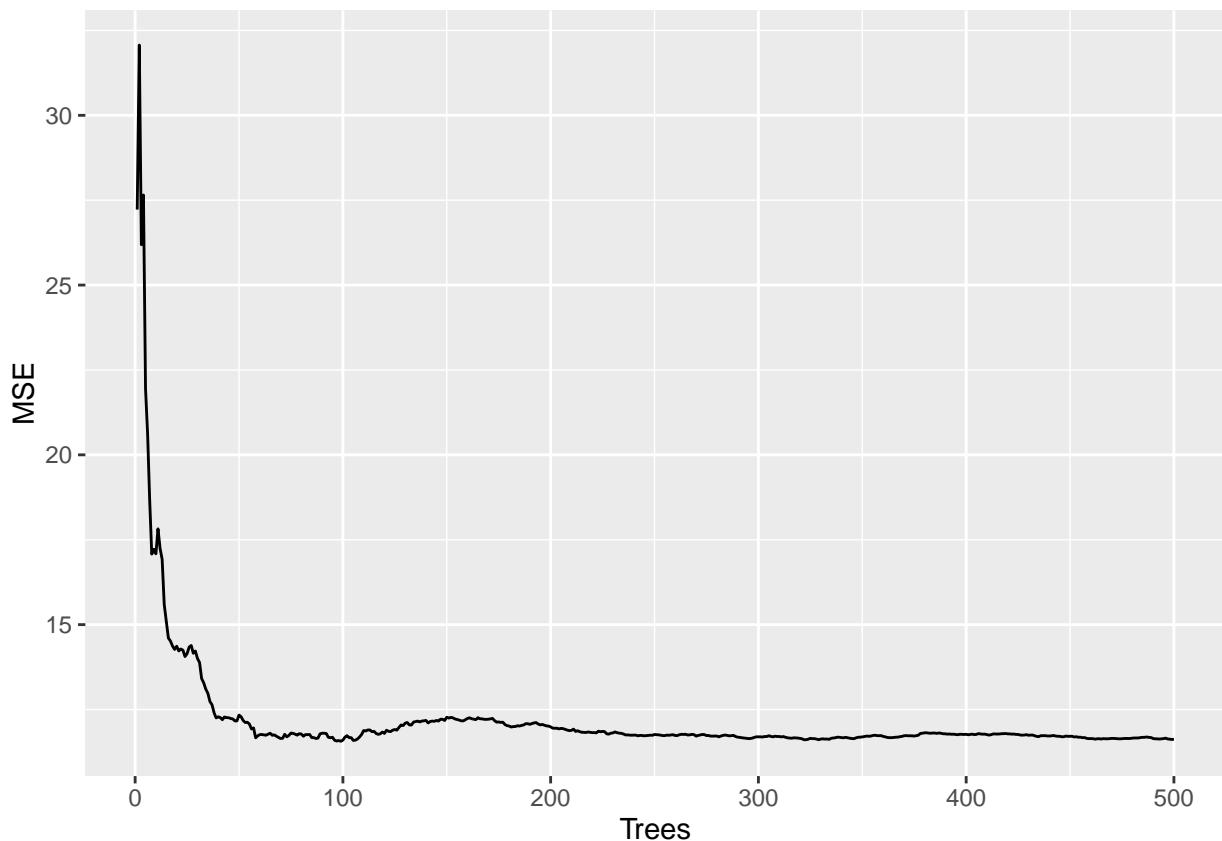
Apartado c)

Ahora vamos a ajustar un RandomForest. Ajustaremos 2 y miraremos cuándo se empieza a estabilizar el error. A partir de ahí veremos dónde se alcanza el mínimo del error.

```

randomFor <- randomForest(medv ~., data = Boston, subset = index, importance = TRUE)
ggplot(data = data.frame(Trees = 1:randomFor$ntree, MSE = randomFor$mse), aes(x = Trees, y = MSE)) +
  geom_line()

```



```
summary(randomFor)
```

```
##                                     Length Class Mode
## call                           5    -none- call
## type                          1    -none- character
## predicted                     404   -none- numeric
## mse                           500   -none- numeric
## rsq                           500   -none- numeric
## oob.times                      404   -none- numeric
## importance                    26    -none- numeric
## importanceSD                  13    -none- numeric
## localImportance                0    -none- NULL
## proximity                     0    -none- NULL
## ntree                          1    -none- numeric
## mtry                          1    -none- numeric
## forest                         11   -none- list
## coefs                         0    -none- NULL
## y                             404   -none- numeric
## test                          0    -none- NULL
## inbag                         0    -none- NULL
## terms                        3    terms  call

randomFor.prediction <- predict(randomFor, newdata = Boston.test)
randomFor.error <- mse(medv[-index], randomFor.prediction)

randomFor.optimalntree <- randomForest(medv ~., data = Boston, subset = index, importance = TRUE,
                                         ntree = which(randomFor$mse == min(randomFor$mse[100:length(randomFor))]))
randomFor.optimalntree.prediction <- predict(randomFor.optimalntree, newdata = Boston.test)
```

```
randomFor.optimalntree.error <- mse(medv[-index], randomFor.optimalntree.prediction)
```

En este caso el error se estabiliza a partir de 100, más o menos. El número de árboles óptimo es 105. El error de test usando 500 es 11.6847429, y el error de test usando el óptimo es 12.0288173.

En este caso usar el óptimo no nos ha ofrecido mejora sobre el error de test. Parece que está sobreajustando.

Apartado d)

Ajustamos un modelo de regresión con Boosting. Como no tiene el mismo número de árboles por defecto que *randomForest*, lo ajustamos nosotros a 500, que es el parámetro por defecto del otro.

```
boosting <- gbm(medv ~ ., data = Boston.train, distribution = "gaussian", n.trees = 500)
boosting.prediction <- predict(boosting, Boston.test, n.trees = 500)
boosting.error <- mse(medv[-index], boosting.prediction)
```

El error de test es 69.6150455.

La diferencia con bagging y *randomForest* es notoria. Boosting se comporta mucho peor, dejando en el mejor lugar a *RandomForest* por muy poquito sobre Bagging. Esto es previsible, ya que *randomForest* tiene una diversificación que no tiene *Bagging*, este tiende a sobreajustar más.

Ejercicio 4

Apartado a)

Cogemos una muestra aleatoria de 800 elementos y lo usamos como training.

```
set.seed(1000000004)
index <- sample(nrow(OJ), 800)
OJ.train <- OJ[index,]
OJ.test <- OJ[-index,]
```

Ajustamos un árbol con la variable *Purchase* como objetivo.

```
model.tree <- tree(Purchase ~ ., data = OJ.train)
```

Apartado b)

Veamos un resumen del árbol.

```
(model.tree.summary <- summary(model.tree))
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"          "SpecialMM"        "ListPriceDiff"   "PctDiscCH"
## Number of terminal nodes:  7
## Residual mean deviance:  0.7454 = 591.1 / 793
## Misclassification error rate: 0.1688 = 135 / 800
```

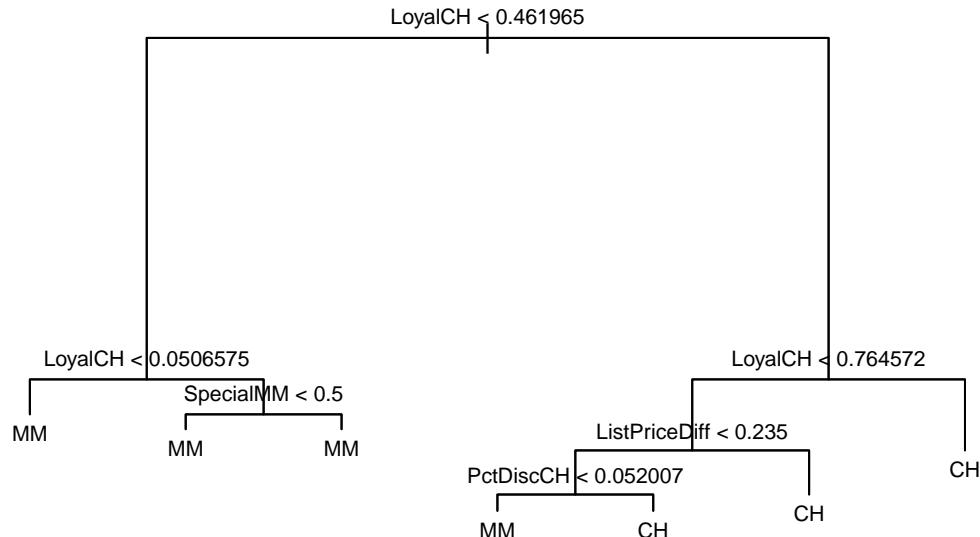
El número de nodos terminales es de 7, y tiene un error del 16.875%. Sin probar otros clasificadores, podemos afirmar que el árbol no es un buen clasificador para este problema, siguiendo la regla de que un clasificador se

considera bueno a partir del 90% de acierto. Las variables usadas han sido LoyalCH, SpecialMM, ListPriceDiff, PctDiscCH.

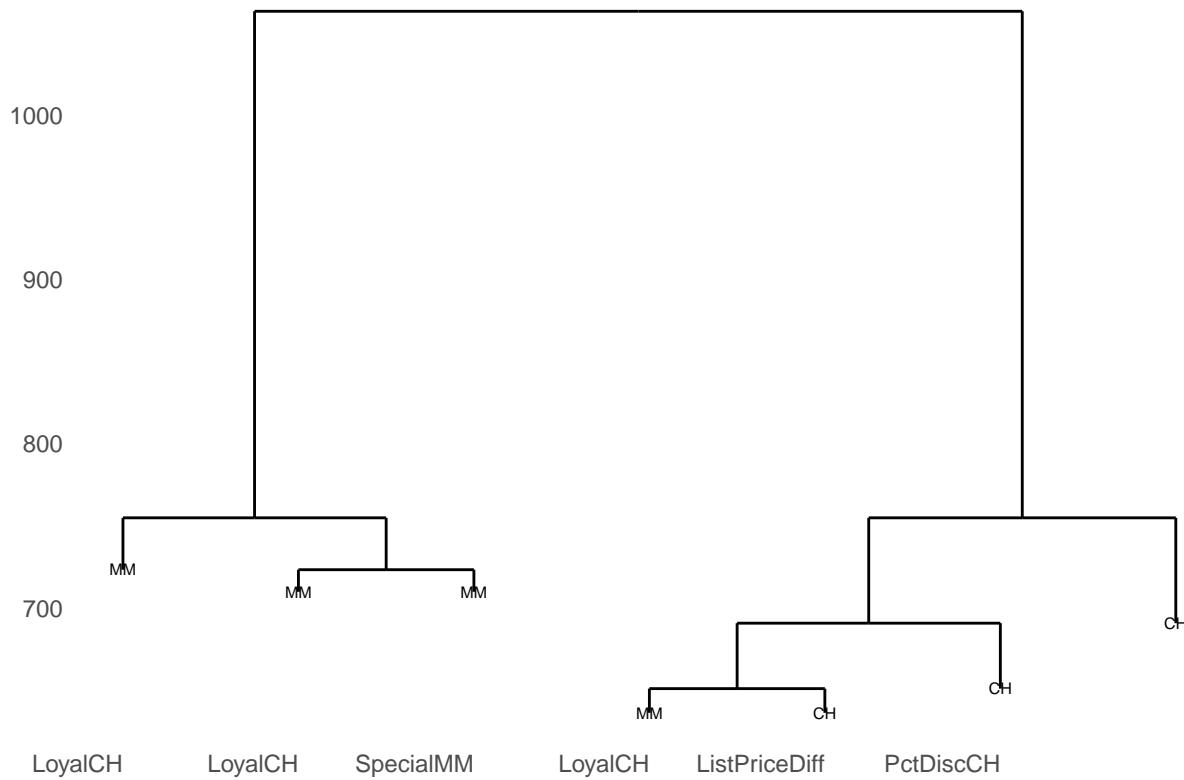
Apartado c)

Dibujamos el árbol obtenido.

```
plot(model.tree, main="Classification tree")
text(model.tree, cex=.7)
```



```
ggdendrogram(model.tree, rotate = FALSE, size = 2)
```



La variable que más ganancia de información tiene es *LoyalCH*, los dos primeros niveles del árbol usan solamente esta variable. Además, hay algunas ramificaciones sin utilidad: Si $LoyalCH < 0.27$, no hace falta volver a distinguir para deducir *MM*, por ejemplo.

Apartado d)

Aplicamos el árbol a nuestros datos de test.

```
prediction.tree <- predict(model.tree, OJ.test, type = "class")
(error.test.tree <- confusionMatrix(prediction.tree, OJ.test$Purchase))

## Confusion Matrix and Statistics
##
##          Reference
## Prediction CH MM
##      CH 129 23
##      MM  29 89
##
##          Accuracy : 0.8074
##                  95% CI : (0.7552, 0.8527)
##      No Information Rate : 0.5852
##      P-Value [Acc > NIR] : 6.519e-15
##
##          Kappa : 0.6064
##  Mcnemar's Test P-Value : 0.4881
##
##          Sensitivity : 0.8165
##          Specificity : 0.7946
##      Pos Pred Value : 0.8487
##      Neg Pred Value : 0.7542
##          Prevalence : 0.5852
##      Detection Rate : 0.4778
##  Detection Prevalence : 0.5630
##      Balanced Accuracy : 0.8055
##
##      'Positive' Class : CH
##
```

La matrix de confusión muestra todos los datos necesarios. El error de test es $1 - Accuracy$, 0.1925926

Apartado e)

Aplicamos la función cv.tree() a los datos de training y veamos qué hace.

```
model.cv.tree <- cv.tree(model.tree, K = 5)
model.cv.tree

## $size
## [1] 7 6 5 4 3 2 1
##
## $dev
## [1] 750.9856 730.3341 728.2452 735.6071 749.3866 773.1692 1063.7738
##
## $k
```

```

## [1]      -Inf  13.43964  14.74170  31.59268  39.85655  64.16091 308.55587
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"        "tree.sequence"

```

Para cada tamaño de árbol, calcula su error con validación cruzada. El mínimo error es el mínimo de *dev*. Lo veremos con una gráfica en el siguiente apartado.

Apartado f) (Bonus-4)

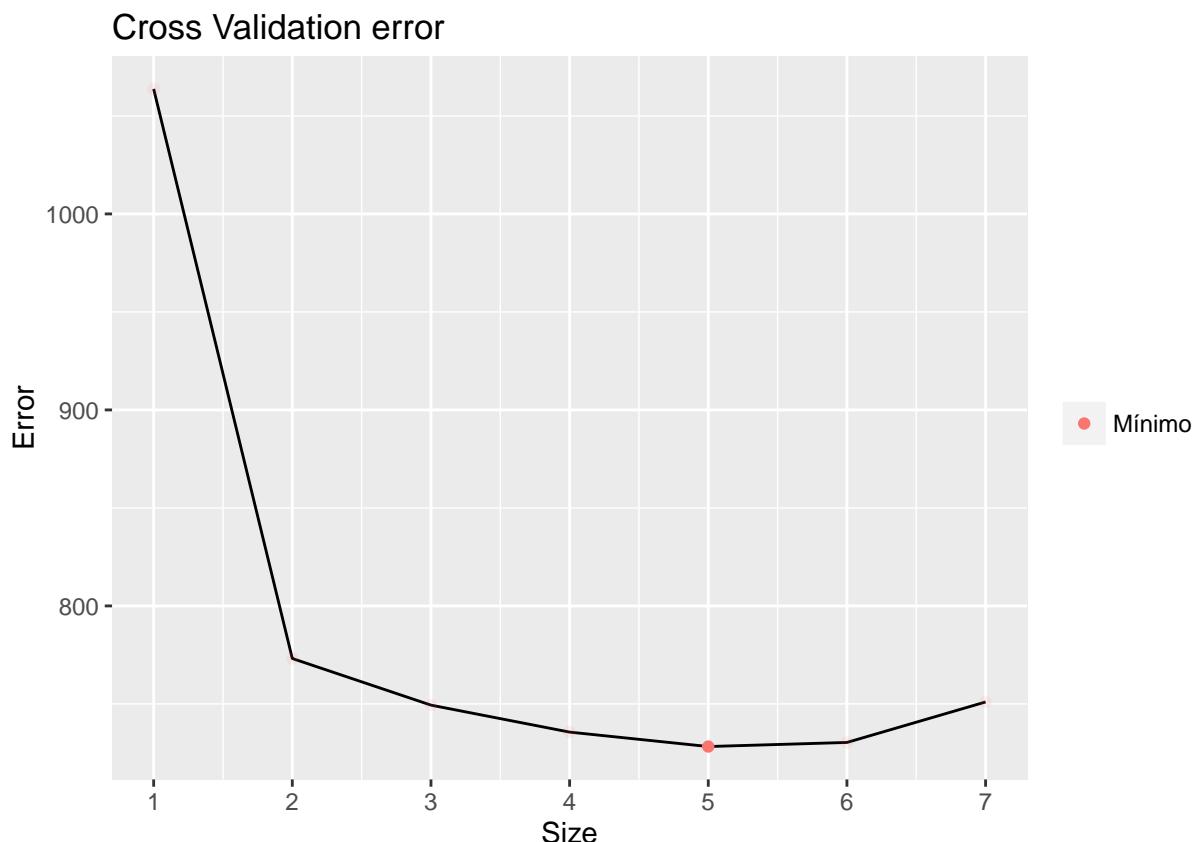
```

error.tree <- data.frame(size = model.cv.tree$size,
                          error = model.cv.tree$dev)

error.tree$alpha <- ifelse(error.tree$error == min(error.tree$error), 1, 0)

ggplot(error.tree, aes(x=size,y=error)) + geom_line() +
  geom_point(aes(colour = "Mínimo", alpha = alpha )) +
  scale_x_continuous(breaks = error.tree$size) +
  guides(alpha=FALSE) +
  theme(legend.title = element_blank()) +
  labs(title = "Cross Validation error",
       x = "Size",
       y = "Error")

```



Podemos ver que el mínimo de error se alcanza en 5.